# The Journal of Functional and Logic Programming

*The MIT Press*

Volume 2000, Article 5
*31 March 2000*

ISSN 1080–5230. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493, USA; (617)253-2889; *journals-orders@mit.edu, journals-info @mit.edu*. Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://www.cs.tu-berlin.de/journal/jflp/*
- *http://mitpress.mit.edu/JFLP/*
- *gopher.mit.edu*
- *ftp://mitpress.mit.edu/pub/JFLP*

*The Journal of Functional and Logic Programming* is a peer-reviewed and electronically published scholarly journal that covers a broad scope of topics from functional and logic programming. In particular, it focuses on the integration of the functional and the logic paradigms as well as their common foundations.



*Electronic Mail:* *jflp.request@ls5.informatik.uni-dortmund.de*

# A Game-based Architecture for Developing Interactive Components in Computational Logic

Kostas Stathis

31 March 2000

**Abstract**

We present a game-based architecture for developing complex interactive components in computational logic. Interactive components are developed either as players making moves according to the rules of a game or as umpires that enforce the rules, thereby controlling the behaviours of players. Centralised organisations based on umpires and decentralised organisations based on autonomous players can then be combined to produce interactive systems of a very complex nature. The potential of the resulting methodology is exemplified by showing how to formulate interaction protocols for software agents in the context of connected community systems.

## 1 Introduction

In [SS96] we introduced the notion of games as a metaphor for developing interactive systems, and in [Sta96] we presented a logic-based framework—motivated by our work in [SS97]—demonstrating how to apply games to build practical applications such as knowledge-based front-ends [Sta94]. Interactive systems of this type require centralised control, and as a result in [Sta96] a game is played via an umpire. The umpire is a component that displays the state of the game, provides means by which user players select moves, enforces compliance of all the players with the rules, and thereby controls

the interactions. The advantages of the resulting framework is that complex interactive systems, such as multilingual front-ends [SS98], can be built systematically from component subgames.

In this paper we revisit and extend the existing formulation of the games metaphor to develop interactions of the type found in multiagent systems [WJ95]. We are motivated by our work in connected community systems [MPS99], where an agent is an autonomous software component that performs tasks on behalf of a user in the electronic environment of a community. In this context the issue becomes how to use games to formulate agent autonomy, where an umpire is not necessary to control the interactions. To address this issue, the contribution of this paper is to provide an alternative organisation of a game, where control is distributed to players. The resulting formulation is an architecture for possible organisations of an interactive system where control is either centralised to an umpire, or distributed to individual players, or both. We exemplify the architecture by showing how to formulate interactions resulting from agent interaction protocols [Fou97].

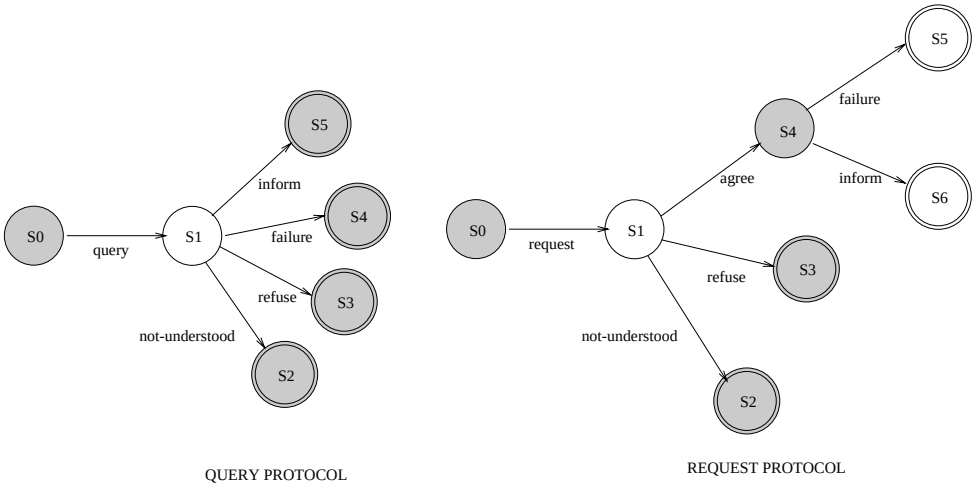


Figure 1: Two interaction protocols

The agent interaction protocols shown in Figure 1 are used as examples throughout the paper. The QUERY protocol illustrates how an agent queries another agent for information, while the REQUEST protocol shows how an agent asks another agent to perform an action on its behalf. We formulate the rules of a protocol as the rules of a game, and the agents using a protocol

to interact as the players playing a game. Shaded circles in Figure 1 represent states where the initiator player can move, while white circles represent states where the other player can move. The possible moves at different states are represented by arrows, while terminating states are represented with double circles.

The paper is structured as follows. We revisit the existing formulation of games in Section 2, where we specify a game as a component with a local state, a set of moves, and a set of rules determining how the state evolves as a result of players making moves. In this new formulation we the show in Section 3 how to play a game via an umpire, a separate component to control the behaviour of players on the state of the game. In Section 4 we show how players may play a game autonomously, without an umpire, but by sharing the rules and making moves in a game through the medium of an environment. Compound games are presented in Section 5, showing how to extend the framework to support complex interactions for umpired and autonomous play. We conclude in Section 6, where we also discuss the differences of this paper with our previous and related work.

# 2 Atomic games

To represent any game, we need to decide how to specify the state, the moves, and the rules. The state contains the properties describing different configurations of the game, including who plays in the game, and possibly containing a history of the moves being made. The moves represent the different types of actions available in the game. The rules describe which moves are legal at different stages of the game, and how these moves affect the state of the game. In this section we present a general specification of the state, the moves, and the rules of a game. The specification is instantiated by showing how to represent the QUERY protocol as a game.

## 2.1 The moves of a game

We represent a move as an Act selected by a Player as follows:

select(Player, Act).

Player and Act are terms whose representation depends on the specific game at hand. For example, the moves for the QUERY protocol are defined as follows:

```
select(Player, query(Query));
select(Player, inform(Query, Answer));
select(Player, failure(Query));
select(Player, refuse(Query));
select(Player, notunderstood(Query)).
```

The first term describes a Player who asks a Query, the second a Player who informs an Answer, the third a Player who fails to answer, the fourth a Player who refuses to answer, and the fifth a Player that does not understand the Query. We assume propositional queries; thus an answer is either yes or no.

## 2.2 The state of a game

We treat the state of a game as a separate component specified by rules of the following form:

```
rule(State, Property, Conditions).
```

Such a rule states that a Property holds in the State of the game if the list of all the Conditions also hold in that state. For example, we can describe an instance of the state for a QUERY protocol as follows:

```
rule(qprotocol(qp1), player(p1), []) ),
rule(qprotocol(qp1), player(p2), []) ),
rule(qprotocol(qp1), about(tel(jim,5946330)), []) ),
rule(qprotocol(qp1), next(p2), []) ).
rule(qprotocol(qp1), last(p1), []) ).
rule(qprotocol(qp1), previous( select(p1, query(tel(jim,5946330), yes))), []) ).
```

The state of the QUERY protocol qprotocol is indexed by identifier qp1. This holds an instance of the game where player p1 has already asked player p2 to check whether the telephone number of jim is 5946330. The state also holds what the protocol is all about (i.e., the query), who played last, and who plays next, and keeps a record of the previous move made.

The properties describing the state of a game are interpreted as goals of the solve/2 metainterpreter [Bro93], as defined below:

```
solve(State, []).
solve(State, [Goal | Goals]):- solveOne(State, Goal), solve(State, Goals).
```

System goals, negation, and rules are interpreted separately as follows:

```
solveOne(State, Goal):- system(Goal), !, Goal.
solveOne(State, not [Goal | Goals]):-!, not solve(State, [Goal | Goals]).
solveOne(State, not Goal):- not solveOne(State, Goal).
solveOne(State, Goal):- rule(State, Goal, Conditions), solve(State, Conditions).
```

Although solve/2 has been extended to handle state changes [BT91], in this paper we use the program effects/2 to apply the effects of moves (or actions) on the state of a game:

```
effects(State, []).
effects(State, [Action | Actions]):-
        effectsOne(State, Action),
        effects(State, Actions).
```

The effects of one Action on the State are specified as follows:

```
effectsOne(State, Action):-
        effectsRule(State, Action, Conditions, Actions),
        solve(State, Conditions),
        effects(State, Actions).
```

Rules of the form

```
effectRule(State, Action, Conditions, Actions)
```

are interpreted as follows: The list of Actions are carried out if the Conditions of the Action are solved on the State of the game. The additional clauses

```
effectsOne(State, add( Property )):- assert( rule(State, Property, []) ).
effectsOne(State, delete( Property )):- retract( rule(State, Property, []) ).
```

simulate assert/1 and retract/1 for a game state described by rule/3. As an example of an effects rule, consider how to formulate the effects of a move that is not a query (i.e., inform, failure, refuse, and notunderstood) on the state of a QUERY protocol game:

```
effectsRule(qprotocol(Id), select(P, Act), [not Act = query(_)],[
        delete(previous(select(L, _))),
        add(previous(select(P, Act))),
        delete(next(P)),
        add(next(L)),
        delete(last(L)),
        add(last(P))]).
```

The effects change what is the previous move made and whose turn it is next.

## 2.3 The rules of a game

The following logic program describes the rules of a game:

```
game(State, Result):-
        terminating(State, Result).
game(State, Result):-
        not terminating(State, _),
        valid(State, Move),
        effects(State, [Move]),
        game(State, Result).
```

The two clauses above are interpreted as follows: If a terminating state is reached, the rules of the game return Result as output. Otherwise, in a nonterminating state the effects of a valid Move are applied to produce a new State. Interaction as a game continues in the new, implicitly updated state until termination is reached. terminating/2 rules are specified as

```
terminating(State, Result):- solve(State, terminating(Result)).
```

For example, we specify how the QUERY protocol terminates with rules of the form

```
rule(qprotocol(Id), terminating( answered(Pa, Pb, Query, Answer) ), [
        next(Pb),
        previous(select(Pa, inform(Query, Answer)))
]).
```

To represent the valid moves, we define a valid move in terms of available and legal moves:

```
valid(State, Move):- available(State, Move), legal(State, Move).
```

Conceptually, available moves are those moves that can be physically selected in a game, while legal moves are those available moves that are allowed at different stages of the game. In programming terms, available/2 definitions act as generators, while legal/2 definitions act as filters.

Practical applications—see, for instance, [Sta96]—require that some legal moves must be forced at specific stages of the game irrespective of what other moves are possible, as if some legal moves overwrite the rest of the legal moves. We express this layering of legal moves by defining them in terms of necessary and possible moves:

```
legal(State, Move):- necessary(State, Move).
legal(State, Move):- not necessary(State, _), possible(State, Move).
```

necessary/2 definitions always overwrite the legal moves described by possible/2 definitions. The moves of a specific game are then defined by specifying the conditions under which moves are available, possible, or necessary and as a result implicitly derive when moves are legal and valid. We define available, necessary, and possible moves as

```
available(State, Move):- solve(State, available(Move)).
necessary(State, Move):- solve(State, necessary(Move)).
possible(State, Move):- solve(State, possible(Move)).
```

In the QUERY protocol it is available to any protocol player to ask a query:

```
rule(qprotocol(Id), available( select(P, query(Query)) ), [player(P)]).
```

It is also necessary at the beginning that the player who plays next asks a query:

```
rule(qprotocol(Id), necessary( select(P, query(Query)) ),[
        next(P),
        not last(_)
]).
```

After a query is asked, possible moves are defined by a rule of the following form:

```
rule(qprotocol(Id), possible( select(Next, Act) ),[
        next(Next),
        previous(select(Last, query(Q))),
        member(Act, [inform(Q, A),failure(Q),refuse(Q),notunderstood(Q)])
]).
```

The full definition of the protocol is given in the appendix.

# 3 Umpired games

Given the rules and the initial state of a game, execution of game/2 describes all possible results of that game. To play a game, however, requires a different mode of execution whereby actual moves made by players effectively select one result from all possible ones. This section shows how to play a game

via an umpire. The umpire is a component that checks moves made by players and presents their effects on a display, in this way controlling their interactions. The resulting representation of the umpire is exemplified by showing how to umpire the QUERY protocol, specified as a game in the previous section.

## 3.1 The umpire

We use the logic program proposed in [Sta98] to define an umpire of a game:

```
umpire(U, S, R, game(S, R)):-
        stop(U, S, R, terminating(S, R)).
umpire(U, S, R, game(S, R)):-
        not terminating(S, _),
        check(U, M, S, valid(S, M)),
        display(U, S, M, effects(S, M)),
        umpire(U, S, R, game(S, R)).
```

The umpire's identifier U and the state of the game S are provided as input when umpire/4 is started, while the result R is the output returned when the program terminates. Moves made by players are extracted and checked for validity by the umpire; only the effects of a valid move M are displayed. The umpire finally obtains the result R and stops the game. Note also that umpire/4 is easily portable. To port the implementation of an umpire to a new platform, one may need to change only stop/4, check/4, and display/4, especially those parts that depend on primitives of the old platform.

## 3.2 Checking move validity

A move is checked in a game by developing further the check/4 predicate. This is defined either in *forward mode*, where all the valid moves are provided to a player to select one, or in *backward mode*, where a player selects a move and the umpire checks that it is valid using the rules. In both modes, we assume that players are components holding selection strategies that are interpreted by solve/2. Then, checking a move in forward mode, we write

```
check(U, select(P, Act), S, Valid):-
        findall(Act, Valid, Acts),
        solve(P, [select(S, Acts, Act)]).
```

When calling check/4, Act is a free variable, but the rest of the variables in the head are instantiated. Note that calling findall/3 does not instantiate Act, but calling the selection strategy of the player P through solve/2 does. Note also that the variable Valid unifies at execution time (see definition of umpire/4) with the head of the rules defining valid moves:

```
valid(S, select(P, Act)).
```

Backward mode is defined slightly differently:

```
check(U, select(P, Act), S, Valid):- solve(P, [select(S, Act)]), Valid.
```

Now, the move is first selected by the player, and then the umpire checks that the move is valid.

## 3.3 Player strategies

To illustrate how to define player strategies, consider two players p1 and p2, playing the QUERY protocol game via an umpire u1 that checks moves in backward mode. Human players make moves via a display using solve/2 (the simplest possible definition is given below):

```
solveOne(P,select(qprotocol(Id),Act)):- human(P),write("Move ?"),read(Act).
```

System players are represented by logic programs. In this umpired variation of play we specify their behaviour with rules. Suppose that p1 is a player represented as a logic program, and that p1 plays first. The selection strategy of p1 in this case asks a query:

```
rule(p1, select(qprotocol(Id), query(Query)), [
        not solve(qprotocol(Id), previous(_)),
        solve(qprotocol(Id), [about(Query)])
]).
```

Note that we now need to allow solve/2 to appear in the body of rules, so that the system player can access the state of the game. This can be accommodated by adding to the definition of solve/2 the following definition:

```
solve(A, solve(B, G)):- solve(B, G).
```

Now suppose that logic program p2 is also a player that has to answer the query. Assuming propositional queries with a yes/no answer, the rule below allows p2 to inform the answer yes:

```
rule(p2, select(qprotocol(Id), inform(Q,yes)), [
        intelligible(Q),
        public(Q),
        solve(qprotocol(Id), [previous(select(P, query(Q))]),
        Q
]).
```

intelligible/1 and public/1 are defined locally for p2. If the query Q cannot be solved in p2's knowledge base, the player informs with a no. Similarly, the rest of the strategy rejects queries asking private questions, while unintelligible queries result in a query not being understood.

## 3.4 Completing the umpire

We can complete the definition of the umpire by showing how the umpire displays the effects of a valid move, and how the umpire stops a game. A valid move is displayed by the following program:

```
display(U, S, M, Effects):- Effects, solve(U, display(S, M)).
```

Interpretation of display rules ensures that the effects of a move are shown on the display of the application. In the context of the QUERY game umpired by the umpire component u1, for example, a display rule is defined as

```
rule(u1, display(qprotocol(Id), M), [write(M), nl]).
```

Similarly, stopping a game is now specified as

```
stop(U, S, R, Terminating):- Terminating, solve(U, stop(S, R)).
```

Again, in the context of the QUERY game, a rule that stops the game is defined as

```
rule(u1, stop(qprotocol(Id), R), [
        writelist(["Result for QUERY protocol:", Id, "is ", R]),
        nl
]).
```

To give the full picture of how an umpired game is started, we also need to show how the initial state of a game is constructed, and how the umpire program is activated. One way to construct the initial of a game is

```
new( qprotocol(Id, First, Second, Query) ):-
        assert( rule(qprotocol(Id), player(First), []) ),
```

```
assert( rule(qprotocol(Id), player(Second), []) ),
assert( rule(qprotocol(Id), next(First), []) ),
assert( rule(qprotocol(Id), about(Query), []) ).
```

For simplicity, we do not specify explicitly the member predicates of a protocol state, but they are required in practice. We are now in a position to start an umpired game. The conjunction

```
?- new(qprotocol(q1,p1,p2,tel(jim,5946330))), umpire(u1,qprotocol(q1),R,G).
```

allows player components p1 and p2 to play the query protocol game G via the umpire component u1 and to return R as the result.

# 4 Autonomous play

An alternative way of using the rules of a game is to play the game with autonomous players, where an umpire is absent. This view is motivated by applications such as multiagent systems, for example, [MPS99], where an agent is an autonomous software component that reacts according to a changing and evolving environment. We develop an agent as an autonomous player of a game, and we introduce the notion of an environment that holds the state of the game and displays the moves made. We exemplify the idea by showing how to communicate with the query protocol using autonomous players, and, by analogy, showing how to use games to develop communication amongst autonomous agents.

## 4.1 The environment of a game

In developing autonomous play, the game is played through the medium of an environment. In the extended game metaphor presented here, an environment is the implementation of a game state. As user players must be supported, the environment is required to display the effects of moves. A move causes the environment to evolve. Assuming a logic-programming platform with a multithreaded capability of the type described in [CC93], we represent the evolution of the environment with the following program:

```
evolve(E, S, R, game(S, R)):-
        halt(E, S, R, terminating(S, R)).
evolve(E, S, R, game(S, R)):-
        not terminating(S, _),
```

```
        consume(E, S, M, available(S, M)),
        display(E, S, M, effects(S, M)),
        evolve(E, S, R, game(S, R)).
```

The environment thread halts when the state of the game has terminated. Otherwise, the environment consumes a move made by a player and displays the effects of the move to the rest of the players, including the one that made the move. Note that the environment consumes only physically available moves; it does not need to know whether moves are valid or legal. Also, a consumed move is not remembered once displayed; that is, the environment is destructive in that it does not support full history explicitly. (It is the responsibility of players to remember what they do in various stages of the game.) To define a specific environment, we need to specify the lower-level implementation rules as

```
halt(E, S, R, Terminating):- solve(E, [Terminating, halt(S, R)]).
consume(E, S, M, Available):- solve(E, [consume(S, M), Available]).
display(E, S, M, Effects):- solve(E, [Effects, display(S, M)]).
```

by defining halt/2, consume/2 and display/2. Examples for some of these predicates are given in Section 4.3. First, however, we show how to represent autonomous players.

## 4.2 Autonomous players

Assuming an evolving environment that displays moves, we represent autonomous players in that environment as active threads represented by the following program:

```
play(P, E, S, R, game(S, R)):-
        exit(P, E, S, R, terminating(S, R)).
play(P, E, S, R, game(S, R)):-
        not solve(P, [terminating(S, _)]),
        choose(P, E, S, M, valid(S, M)),
        execute(P, E, S, M, effects(S, M)),
        play(P, E, S, R, game(S, R)).
```

A player P concludes a game by exiting a terminating state S in an environment E. Otherwise, the player chooses a move M in the environment E, executes it in E, and continues the play until the game ends. For any player P we use solve/2 to interpret the terminating conditions and the validity of

moves. The clauses below are based on the definition of solve/2, provided earlier, and assume that the rules of a game are defined once and shared by all players as

```
solveOne(P, terminating(S, R)):- solve(S, [terminating(R)]).
solveOne(P, valid(S, R)):- valid(S, M).
solveOne(P, effects(S, M)):- effects(S, M).
```

In general, players may hold different, and possibly conflicting, interpretations of rules. Determining how to accommodate this type of player is, however, beyond the scope of this work.

Based on the representation of an autonomous player as above, we represent the lower-level implementation predicates as

```
exit(P, E, S, R, Terminating):- solve(P, [Terminating, exit(E, S, R)]).
choose(P, E, S, M, Valid):- solve(P, [choose(E, S, M), Valid]).
execute(P, E, S, M, Effects):- solve(P, [Effects, execute(E, S, M)]).
```

In other words, to completely define a player, we need to define exit/3, choose/3, and execute/3 in the local database of each player. We show next how to instantiate these rules in the context of an environment where two system players play the QUERY protocol.

## 4.3 Synchronisation

The issue of developing autonomous players interacting through an environment is how players synchronise their moves with other players and the environment. We sketch how synchronisation is achieved in this context by illustrating how two agents may synchonise their moves when they play the QUERY protocol. We capitalise on the fact that turn-taking is described in the rules of the game. We assume that the environment buffers move before they are consumed. We also assume that the effects of these moves are applied at the environment level by the display.

### 4.3.1 The environment definition

Players in autonomous play share two assertions in the database of the environment:

- moved/0 indicates that a player has made a move so that the environment can consume it;

• canmove/0 indicates that a consumed move has been displayed by the environment, so that another player can make a move.

Initially moved/0 is not in the environment, but canmove/0 is. This means that when starting a game, the environment is waiting for a move to be made in order to consume it, and that a player can move. Assuming an environment instance e1, we define how a move is consumed:

```
rule(e1, consume(S, M), [wait(moved), getbuffer(S, M)]).
```

First, the environment waits for a player to make a move; in other words, the player adds moved in the state of the environment. wait/1 is like a primitive that is either supported by the implementation platform or specified as follows:

```
solve(C, wait(G)):- solve(C, G),!.
solve(C, wait(G)):- solve(C, wait(G)).
```

Once such a move is made, the environment gets the move from the buffer of moves for the game being played. The predicate getbuffer/2 allows the environment to access a move that has to be consumed; its implementation is platform specific. Once a move is consumed (i.e., deleted from the buffer), the environment displays that move using an effect rule:

```
effectsRule(e1, display(S, M),
        [writelistnl(["In : ", S, "move made is : ", M])],
        [delete(moved),
        add(canmove)
]).
```

This simple definition allows the environment to realise that a move is made by a player in the next cycle, while a player whose turn is next can now move.

### 4.3.2 Synchronising players

Before a player actually selects a move, this player needs to synchronise with the other players playing the game. This is expressed by the following rule:

```
rule(p1, choose(E, S, M), [synchronise(E,S), select(S, M)]).
```

For a player p1 playing the QUERY protocol, we define synchronisation as an effects rule:

```
effectsRule(p1, synchronise(E,S),
        [self(Me),
```

```
        solve(S, wait(next(Me))),
        solve(E, wait(canmove))],
        [effects(E, [delete(canmove)])
]).
```

To interpret a predicate like synchronise/2, we need to include the following definitions for solve/2:

```
solve(S, G):- effects(S, G).
solve(P, self(P)).
```

The first definition allows goals that cannot be solved with rules to be solved with effect rules; this is the case with synchronise/2. The second definition allows a component to access itself. We also need to add

```
effects(C, effects(D, G)):- effects(D, G).
```

to allow the effects procedure of a component to apply the effects of a rule on another component, for example, how a player may affect the environment.

We are now in a position to define how a move is executed using an effect rule:

```
effectsRule(p1, execute(E,S,M),
        [buffer(S, M)],
        [effects(E, add(moved))
]).
```

First the player makes sure that the move is buffered, and then the player changes the environment that it has moved.

### 4.3.3 Other synchronisation techniques

The assertions moved/0 and canmove/0 act as *shared variables* resembling Dijkstra's P and V variables to handle mutual exclusion and to allow player and environment threads to synchronise. Another way to accomplish synchronisation is to use *communication and message passing*. To achieve this type of synchronisation, we introduce the *synchronisation game*, which supports the communication required for the players to synchronise their moves. Playing two games simultaneously introduces the issue of compound games, which we discuss next.

# 5 Compound games

Compound games are complex games composed from simpler, possibly atomic, subgames. An example of a compound game is a master's game of chess, where a chess master plays several chess games with different student players simultaneously. Based on our previous work in applying compound games to develop interactive systems [Sta96], in this section we show how to develop compound games for interactive components in the context of umpired and autonomous play.

## 5.1 Playing multiple protocols via a compound game

To illustrate how to specify the interactions of a compound game, we consider a game that allows an interactive component to start multiple QUERY protocols with other components at the same time. A participant is also allowed to suspend a protocol in favour of playing another one, and resume a protocol that has been suspended. We identify the state of such a game with the term

multiple(Id).

For simplicity of presentation, this game has three moves:

- start(Id, P, Q) starts a new QUERY game identified by Id with player P about query Q;
- suspend(Id) suspends a game identified by Id;
- resume(Id) resumes a game identified by Id.

For lack of space, we assume a definition of these moves exclusive to the game, similar to the one provided in Section 2.3. In addition, as compound games require submoves to hold moves made in subgames, we use the term

inside(SubS, Move)

to represent a submove. For instance, the term

inside(qprorocol(qp1), select(p1, inform(tel(jim,5946330), yes)))

represents a submove made in a QUERY subgame qp1, where player p1 has selected an inform. To record subgames in the state of the compound game, we use the predicates subgames/1 and suspended/1 as placeholders for current and suspended subgames, respectively.

## 5.2 Specification of coordination in compound games

The main issue in specifying a compound game is how to specify the *coordination* of interactions between component subgames. We deal with this issue by defining the conditions under which a subgame becomes active. We then augment the definition of valid moves presented in Section 2.3 with the following rule:

```
valid(S, inside(SubS, Move)):- active(S, SubS), valid(SubS, Move).
```

The valid moves of a compound game S include the valid moves exclusive to the compound game and the valid moves in the active component subgames with state SubS. Different forms of coordination are possible and can be given by suitable definitions of what an active subgame is. We distinguish two classes of coordination: *free interleaving* and *constrained interleaving*. In any interleaving mode, we represent the state of a compound game as follows:

```
S interleave M.
```

The mode M can be either free or constr, distinguishing in this way between free and constraint interleaving, respectively. In both cases interleave is a Prolog operator. We can now represent free interleaving as

```
active(S interleave free, SubS):- running(S, SubS).
```

We now have to specify locally to each compound game the conditions under which moves from subgames can be selected freely by players of the running subgames. Similarly, constrained interleaving is defined as follows:

```
active(S interleave constr, SubS):- overtaking(S, SubS).
active(S interleave constr, SubS):- not overtaking(S, _), running(S, SubS).
```

This last definition of active/2 relies on domain-specific constraints defining overtaking subgames. These are subgames that, at specific stages of the interaction, have higher priority than the rest of the running games. For both types of interleaving, we define running and overtaking subgames locally to each compound game as

```
running(S, SubS):- solve(S, running(SubS)).
overtaking(S, SubS):- solve(S, overtaking(SubS)).
```

To give an example, consider how we specify overtaking and running rules for the multiple QUERY game. A subgame overtakes another game if it has just been started, that is, when no previous move has been made in it:

```
rule(multiple(Id),
        overtaking(G), [
        subgames(Gs),
        member(G, Gs),
        not solve(G, [previous(_)])
]).
```

Otherwise, any running subgame can be chosen as

```
rule(multiple(Id), running(G), [subgames(Gs), member(G, Gs)]).
```

Both kinds of coordination can share the running game definition above. Both kinds of coordination also use the same definition for effects of submoves:

```
effects(S interleave _, inside(SubS,M)):-
        effects(SubS, [M]),
        effects(S, inside(SubS,M)).
```

The effects of the submove are applied in the state of the subgame first. If the subgame has terminated, we also need to remove the subgame from the list of active subgames. For this reason, the effects of the submove are applied in the state of the compound game (a submove is also a move in the compound game). The definition of a compound game is completed by specifying how the game has terminated; this definition is game specific and is defined similarly to the ones given in Section 2.3.

## 5.3 Umpired compound games

In developing umpires for compound games, we introduce subumpires to umpire subgames. We assume that the umpire of a compound game records the names of subumpires as

```
subumpire(SubS, SubU)
```

meaning that each subgame SubS is umpired by the subumpire SubU. In this way an umpire can dispatch compound moves to the right (sub-) umpire. This organisation also allows that all subgames are managed by the same umpire. How subgames are umpired is specified when a compound is started.

One main issue in umpiring a compound game concerns the way we check moves with subumpires of subgames, which for any interleaving mode is defined as

```
check(U, S, inside(SubS, M), Valid ):-
        trigger(U, SubU, S, SubS, active(S, SubS) ),
        check(SubU, SubS, M, valid(SubS, M) ).
```

A compound move is managed by accessing the subumpire of the subgame and then dispatching it to the right subumpire. The triggering of a subgame is defined for free interleaving as

```
trigger(U, SubU, S interleave free, SubS, active(S interleave free, SubS)):-
        pick(U, SubU, S, SubS, running(S, SubS)).
```

The predicate pick/4 is defined specifically to deal with the way a running game is accessed during the umpiring process. Similarly, for constraint interleaving we write

```
trigger(U, SubU, S interleave constr, SubS, active(S interleave constr, SubS)):-
        accept(U, SubU, S, SubS, overtaking(S, SubS)).
trigger(U, SubU, S interleave constr, SubS, active(S interleave constr, SubS)):-
        not overtaking(S, _),
        pick(U, SubU, S, SubS, running(S, SubS)).
```

The implementation predicate accept/4 is defined to access the subumpire of overtaking subgames. We rely on the following definitions:

```
pick(U, SubU, S, SubS, Running):- solve(U, [pick(SubU, S, SubS), Running]).
accept(U, S, SubS, Overtaking):- solve(U, [accept(SubU, S, SubS), Overtaking]).
```

This requires that we specify pick/3 and accept/3 rules in the database of the umpire. We also need to define

```
solveOne(C, running(S, SubS)):- running(S, SubS).
solveOne(C, overtaking(S, SubS)):- overtaking(S, SubS).
```

so that the umpire can test if a game is running or overtaking.

Another issue of implementing a compound game is how to display a compound move:

```
display(U, S, inside(SubS, M), effects(S, inside(SubS, M))):-
        solve(U, subumpire(SubS, SubU)),
        display(SubU, SubS, M, effects(SubS, M)),
        display(U, S, inside(SubS,M), effects(S, inside(SubS, M))).
```

In any interleaving mode, the effects of a submove are displayed first with the right subumpire, and then we display the effects of the submove at the compound game as a whole.

## 5.4 Autonomous play for compound games

This section is based on the synchronisation techniques used in Section 4.3 and the example definitions for the QUERY protocol given so far.

### 5.4.1 Compound environments

For autonomous play of compound games, we introduce compound environments consisting of subenvironments. The issue now is how the moves are consumed in the subenvironments. We assume that each environment holds a record of the subenvironments as

```
subenv(SubS, SubE).
```

Such an assertion defines that the subgame SubS is played in the subenvironment SubE. With this organisation in mind, the program below shows how to consume a compound move:

```
consume(E, S, inside(SubS, M), available(S, inside(SubS, M))):-
        explore(E, SubE, S, SubS, running(S, SubS)),
        consume(SubE, SubS, M, available(SubS, M)).
```

The environment must explore the various subenvironments to find a move being made by a player. Subenvironments are explored until a subenvironment holds an atomic game with a move made in it. Such an atomic move is consumed by the method defined in Section 4.3.1. Once a subenvironment has been chosen, we need to check that the game it contains is a running game:

```
explore(E,SubE,S,SubS,Running):- solve(E, [explore(SubE,S,SubS), Running])).
```

Domain-specific definitions for explore/3 are needed locally for each (sub-) environment. A consumed move is then displayed. Displaying a compound move is similar to the definition of display/4 given for the compound umpire in the previous section; however, the difference now is that the display is managed by the environment and its subenvironments as opposed to the umpire and its subumpires.

### 5.4.2 Autonomous players for compound games

An autonomous player engaged in a compound game must be able to coordinate the choice and execution of submoves made in the subenvironment

holding the subgames of the main game. We express the choice of a compound move by expressing a preference in a subgame played in a subenvironment, and then by making a move in that subgame:

```
choose(P, E, S, inside(SubS, M), valid(S, inside(SubS, M))):-
        prefer(P, E, SubE, S, SubS, active(S, SubS)),
        choose(P, SubE, SubS, M, valid(SubS, M)).
```

One way to define how a preference is made for subgame is for a player to express it first, and then to check whether this game is active:

```
prefer(P, E, SubE, S, SubS, Active):- solve(P, [prefer(SubE, S, SubS), Active]).
```

By expressing preferences in this way, a player is able to coordinate selections of moves made in multiple subgames.

Once a compound move is selected, it must be executed. This is done by using the following program:

```
execute(P, E, S, inside(SubS, M), effects(S, inside(SubS,M))):-
        solve(E, subenv(SubS, SubE)),
        execute(P, SubE, SubS, M, effects(SubS, [M])),
        execute(P, E, S, inside(SubS,M), effects(S, inside(SubS, M))).
```

First the player executes the move in the subenvironment of the subgame, and then the effects of the same move are applied in the main game to handle any dependencies existing between the subgame and the main game.

# 6 Conclusion

We have presented a game-based architecture for developing interactive components in computational logic. The architecture was illustrated with examples from agent-interaction protocols. These protocols were formulated using two different game organisations. One is based on an umpire component that centrally controls the moves made by player components. The other is based on autonomous players making moves through the medium of an environment holding the state of the game. Both umpired and autonomous play are exemplified also in the context of compound games. The advantage of supporting this class of games is that decentralised and centralised organizations of components can be combined to capture interactions of a complex nature.

One advantage of our approach is that by viewing an agent as an autonomous player of a compound game, this agent can be engaged in multiple

conversations based on interaction protocols. In this context we showed how agents coordinate their conversation acts [BF95] by showing how to coordinate moves in a compound game. We also sketched how synchronisation can be achieved in our framework. In particular, we showed how to simulate synchronisation via shared variables with assertions in a shared database of an environment. Alternatively, synchronisation via communication and message passing can be incorporated by introducing a synchronisation game, supported through the notion of compound games.

From the development point of view, the architecture provides a set of reusable programs that integrate the different components of an interactive system. A component is represented by classes of programs illustrating how instances of these classes can interact with each other. In this context our framework provides the generic part of an application that can be specialised by the developer to support the functionality required from a given application domain. The framework further distinguishes between the notions of compound interactions, coordination of concurrency, and synchronization. Such notions are studied in interactive systems using object-oriented [NGT92, MN98, Joh97, Sch95, SFJ96] or agent-oriented [SC99] techniques. In this case, our work is intended as a study investigating how to support similar techniques in computational logic setting.

Compared with our previous formulation of games using object-level programs [SS97, Sta96], this work introduces metaprograms to interpret the object-level rules that represent games, players, umpires, and environments. The representation chosen in this work is also simpler than the one we present in [Sta98], in that here we reduce the different types of rules that describe a component into two: rules representing how inferences are drawn from existing state properties of a component, and effects rules representing how updates are carried out in that component. The advantages of this approach are that representing a game component is now simpler and the formulating interactions amongst these components is conceptually clearer.

Our work on games bears no relation with *game theory* for economic modelling [VM44] and its applications to interactive systems in the sense presented by Genesereth, Ginsberg, and Rosenschein [GGR86]. Also, our use of games is different from the way games are often applied to model interaction in *theoretical computer science* (as in Abramsky and Jagadeesan [AJ94], Abramsky and McCusker [AM95], and Nerode and Yakhnis [NY92]). Instead, we use games to view interaction as a rule-governed activity with the emphasis more on the existence of clearly specified legal moves rather than

on any specific notion of *winning*. However, notions such as winning are becoming increasingly important in multiagent systems applications, where agents may, for example, negotiate according to the rules of a protocol. The detailed discussion of such issues we intend to present in a future work, where we also plan to investigate the semantics of game interactions.

# Appendix: The QUERY protocol as a game

```
new( qprotocol(Id, PlayerA, PlayerB, Query) ):-
    assert( rule(qprotocol(Id), player(PlayerA), []) ),
    assert( rule(qprotocol(Id), player(PlayerB), []) ),
    assert( rule(qprotocol(Id), next(PlayerA), []) ),
    assert( rule(qprotocol(Id), about(Query), []) ).

rule(qprotocol(Id),
     terminating( answered(Pa, Pb, Query, Answer) ), [
     next(Pb),
     previous(select(Pa, inform(Query, Answer)))
]).
rule(qprotocol(Id),
     terminating( failed(Pa, Pb, Query) ), [
     next(Pb),
     previous(select(Pa, failure(Query)))
]).
rule(qprotocol(Id),
     terminating( refused(Pa, Pb, Query) ), [
     next(Pb),
     previous(select(Pa, refuse(Query)))
]).
rule(qprotocol(Id),
     terminating( didnotunderstand(Pa, Pb, Query) ), [
     next(Pb),
     previous(select(Pa, notunderstood(Query)))
]).

rule(qprotocol(Id), available( select(P, query(Q)) ), [player(P)]).
rule(qprotocol(Id), available( select(P, inform(Q,A)) ), [player(P)]).
```

```
rule(qprotocol(Id), available( select(P, failure(Q)) ), [player(P)]).
rule(qprotocol(Id), available( select(P, refuse(Query)) ), [player(P)]).
rule(qprotocol(Id), available( select(P, notunderstood(Q)) ), [player(P)]).

rule(qprotocol(Id),
     necessary( select(P, query(Query)) ), [
     next(P),
     not previous(_)
]).

rule(qprotocol(Id),
     possible( select(NextP, Act) ), [
     next(NextP),
     previous(select(PreviousP, query(Q))),
     member(Act, [inform(Q,A),failure(Q),refuse(Q),notunderstood(Q)]),
]).

effectsRule(qprotocol(Id),
     select(P, Act),
     [Act = query(Q),
     player(LastP),
     not P == LastP],
     [add(previous(select(P, Act)),
     delete(next(_)),
     add(next(PreviousP)),
     add(last(P))
]).

effectsRule(qprotocol(Id),
     select(P, Act),
     [not Act = query(Q)],
     [delete(previous(_)),
     add(previous(select(P, Act)),
     delete(next(_)),
     delete(last(LastP)),
     add(next(LastP)),
     add(last(P))
]).
```

## Acknowledgements

The comments of two anonymous referees on a previous version of this paper are gratefully acknowledged.

## References

[AJ94] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. *Journal of Symbolic Logic*, 59(2):543–574, June 1994.

[AM95] S. Abramsky and G. McCusker. Games and full abstraction for the lazy $\lambda$-calculus. In *Proceedings of the Tenth IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1995.

[BF95] M. Barbuceanu and S. Fox. COOL: A language for describing coordination in multi-agent systems. In *Proceedings of the First International Conference on Multi-agent Systems*. AAAI Press, 1995.

[Bro93] A. Brogi. *Program Construction in Computational Logic*. Ph.D. thesis, Department of Computing, University of Pisa, Corso Italia 40, Pisa, Italy, 1993.

[BT91] A. Brogi and F. Turini. Metalogic for knowledge representation. In J. Allen, R. Fikes, and E. Sandwell, editors, *Proceedings of Knowledge Representation '91*. Cambridge, Massachusets, 1991.

[CC93] D. Chu and K. Clark. IC-Prolog II: A multi-threaded Prolog system. In G. Succi and G. Colla, editors, *Proceedings of the ICLP'93 Post Conference Workshop on Concurrent, Distributed and Parallel Implementations of Logic Programming Systems*. http://researchsmp2.cc.vt.edu/DB/db/conf/iclp/iclp93-w2.html, 1993.

[Fou97] Foundation for Intelligent Physical Agents, http://drogo.cselt.stet.it/fipa/. *FIPA Specification Part 2: Agent Communication Language*, November 1997.

[GGR86] M. R. Genesereth, M. L. Ginsberg, and J. S. Rosenschein. Cooperation without communication. In *Proceedings of the National Conference on Artificial Intelligence*, pages 51–57. AAAI Press, August 1986.

[Joh97] R. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[MN98] T. D. Meijler and O. Nierstrasz. Beyond objects: Components. In P. Papazoglou and G. Schlageter, editors, *Cooperative Information Systems: Trends and Directions*. Academic Press, 1998.

[MPS99] E. Mamdani, J. V. Pitt, and K. Stathis. Connected communities from the standpoint of multi-agent systems. *New Generation Computing*, 17(4):381–393, August 1999.

[NGT92] O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, 1992.

[NY92] A. Nerode and A. Yakhnis. Modelling hybrid systems as games. In *Proceedings of the Thirty-First IEEE Conference on Decision and Control*, pages 2947–2952. IEEE Computer Society Press, 1992.

[SC99] N. Skarmeas and K. Clark. Content-based routing as the basis for intra-agent communication. In J. P. Muller, M. P. Singh, and A. Rao, editors, *Intelligent Agents V*. Springer-Verlag, Heidelberg, 1999.

[Sch95] D. C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, October 1995.

[SFJ96] D. C. Schmidt, M. Fayad, and R. E. Johnson. Software patterns. *Communications of the ACM*, 39(10):37–39, October 1996.

[SS96] K. Stathis and M. J. Sergot. Games as a metaphor for interactive systems. In M. A. Sasse, R. J. Cunningham, and R. L. Winder, editors, *People and Computers XI (Proceedings of HCI'96)*, BCS Conference Series, pages 19–33, London, August 1996. Springer-Verlag.

[SS97] K. Stathis and M. J. Sergot. Knowledge-based front-ends as games. *Failures and Lessons Learned in Information Systems Management*, 2(1):135–147, 1997.

[SS98] K. Stathis and M. J. Sergot. An abstract framework for globalising interactive systems. *Interacting with Computers*, 9(4):401–416, 1998.

[Sta94] K. Stathis. A FAST front end application. In L. Sterling, editor, *Proceedings of the Second International Conference on the Practical Applications of Prolog*, pages 537–548, London, April 1994. Prolog Management Group.

[Sta96] K. Stathis. *Game-based Development of Interactive Systems*. Ph.D. thesis, Department of Computing, Imperial College, London, November 1996.

[Sta98] K. Stathis. Towards a game-based architecture for developing interactive components in computational logic. In A. Brogi and P. Hill, editors, *Proceedings of the First International Workshop on Component-based Software Development in Computational Logic (COCL98)*, Pisa, Italy, September 1998. http://www.di.unipi.it/ brogi/cocl.html.

[VM44] J. VonNeumann and O. Morgenstern. *Theory of Games and Economic Behaviour*. Princeton University Press, Princeton, 1944.

[WJ95] M. Wooldridge and N. Jennings. Intelligents agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.